Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#22: Proper Use of Dynamic Segments

Rewritten by: Eric Soldan & Andy Stadler September 1990
Written by: Guillermo Ortiz October 1987

This Technical Note discusses strategies that applications can use to deal with dynamic segments.

Changes since November 1988: Rewrote from scratch to address current problems.

When reading the documentation on dynamic segments, it initially appears that they are even **better** than sliced bread. While they are incredibly useful, there are two issues that make dealing with them somewhat tricky. The first involves loading a dynamic segment; the second involves unloading a dynamic segment. Everything else works fine.

Loading Dynamic Segments

Loading dynamic segments is supposed to happen automatically. You are supposed to be able to call the code in the dynamic segment, and the system automatically loads it. As long as there is enough RAM to load the segment, this is exactly what happens.

The problem arises when there isn't enough memory. Immediately you have a number of questions, such as "How do I know if it didn't load?" and "How is the not-enough-memory error returned?" Unfortunately, neither of these questions is applicable. Instead, you get a Fatal System Error, which is not the most useful thing that could happen.

However, there are some reasons for this error. For example, in the Pascal or Toolbox stack frame system, the called function is responsible for removing the parameters pushed onto the stack. If the dynamic segment did not load, these parameters cannot be pulled from the stack, and if they are not pulled from the stack, the operating system cannot return to the caller.

Due to this problem, the best thing to do is to try to load the dynamic segment with `LoadSegName`. If it loads, then there is (obviously) enough RAM for it. If it does not load, then there was not enough RAM; it's that simple. So, to call a function named `dynFN` in a dynamic segment called `dynSeg`, you would do the following:

```
LoadSegName("\pDynSeg");
if (!_toolErr) {
    dynFN(some, number, of, parameters);
    UnLoadSeg(dynFN);
```

```
        }
        else ErrorAlert("\pOut of RAM.");
```

Unloading Dynamic Segments

`UnLoadSeg` used to have a problem, so the above technique would not have worked. As of System Software 5.0.3, this problem has been fixed. In the example, the code `UnLoadSeg(dynFN)` does not pass the address of the `dynFN` that was loaded into RAM. Instead, that address represents the entry in the dynamic segment jump table for that particular function. The jump table is always in RAM. So, you are not actually passing an address of the segment to be unloaded, but an address in the jump table.

The loader is responsible for figuring out that the address is actually an address in the jump table, and it is supposed to unload the segment to which the jump table entry refers. The loader did not handle this case properly until 5.0.3. So, for system disks prior to System Disk 5.0.3, you can preserve the segment number returned by the `LoadSegName` call to issue an `UnLoadSegNum` call to dispose of the dynamic segment. Due to `UnLoadSeg` not doing the job prior to 5.0.3, you could use `UnLoadSegNum`. This also has problems. ExpressLoad changes the segment numbers, so it is difficult to maintain the segment numbers if you change the link script. For these reasons, the below technique should be used for system disks prior to 5.0.3:

```
void   sample()
{
       struct    LoadSegNameOut    dynSegInfo;

       dynSegInfo = LoadSegName("\pDynSeg");
       if (!_toolErr) {
           dynFN(some, number, of, parameters);
           UnLoadSegNum(dynSegInfo.segNum);
       }
       else ErrorAlert("\pOut of RAM.");
}
```

Dynamic Segment Interdependencies: Just Say No

Dynamic Segments calling each other almost **always** lead to unloading conflicts, and more importantly, they defeat the purpose (if they both have to be in simultaneously then they might as well be static). Figure 1 is a sample program layout you may want to consider when designing your application dynamic segment usage:

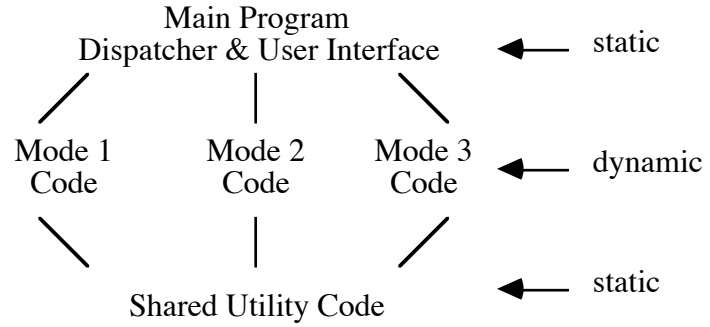

Figure 1–Sample Program Layout

Also, if one of the dynamic segments described is much more than, say, 32K or 40K, you may wish to load a pair (or more) of dynamic segments. These dynamic segment pairs would always be loaded and unloaded simultaneously. Why? Because loading two 25K segments is more likely to succeed than loading one 50K segment.

A Final Warning:

Data in a dynamic segment is a tricky issue. When you call a dynamic segment, you are not sure if it got loaded, or if it was already in RAM, and therefore you cannot be sure of the values in your global data. For example, say that you have a global variable that represents the number of times that you call the dynamic segment. Every time you call the segment, you would increment this variable. This technique works great until the dynamic segment gets purged. Once it is purged, the next time you call it, the variable area would be loaded from disk again, with its original initial value. The count is no longer valid. To fix this, you can place the global could variable in the static globals space for the main code. Then the variable would not get purged, and your count would be valid. Of course, if you have global data that does not ever change, then it is okay for the data to be in the global segment.

Further Reference

- *GS/OS Reference*
- *Apple IIGS Programmer's Workshop Assembler Reference*